Announcements

- Quiz 4
  - Monday
  - On strings

Using Pointers for Array Processing

- Pointer arithmetic allows us to visit the elements of an array by repeatedly incrementing a pointer variable.
- A loop that sums the elements of an array a:
  ```c
  #define N 10
  int a[N], sum, *p;
  ...
  sum = 0;
  for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
  ```

Combining the * and ++ Operators

- C programmers often combine the * (indirection) and ++ operators.
- A statement that modifies an array element and then advances to the next element:
  ```c
  a[i++] = j;
  ```
- The corresponding pointer version:
  ```c
  *p++ = j;
  ```
- Because the postfix version of ++ takes precedence over *, the compiler sees this as
  ```c
  *(p++) = j;
  ```

Using an Array Name as a Pointer

- Suppose that a is declared as follows:
  ```c
  int a[10];
  ```
- Examples of using a as a pointer:
  ```c
  *a = 7;  /* stores 7 in a[0] */
  *(a+1) = 12;  /* stores 12 in a[1] */
  ```
- In general, a + i is the same as &a[i].
  - Both represent a pointer to element i of a.
- Also, *(a+i) is equivalent to a[i].
  - Both represent element i itself.
Using an Array Name as a Pointer

- The fact that an array name can serve as a pointer makes it easier to write loops that step through an array.
- Original loop:
  ```c
  for (p = &a[0]; p < &a[N]; p++)
    sum += *p;
  ```
- Simplified version:
  ```c
  for (p = a; p < a + N; p++)
    sum += *p;
  ```

Using an Array Name as a Pointer

- Although an array name can be used as a pointer, it's not possible to assign it a new value.
- Attempting to make it point elsewhere is an error:
  ```c
  while (*a != 0)
    a++;           /*** WRONG ***/
  ```
- Copy `a` into a pointer variable, then change the pointer variable:
  ```c
  p = a;
  while (*p != 0)
    p++;
  ```

Array Arguments

- When passed to a function, an array name is treated as a pointer.
- Example:
  ```c
  int find_largest(int a[], int n)
  {
    int i, max;
    max = a[0];
    for (i = 1; i < n; i++)
      if (a[i] > max)
        max = a[i];
    return max;
  }
  ```

Array Arguments

- A call of `find_largest`:
  ```c
  largest = find_largest(b, N);
  ```
  This call causes a pointer to the first element of `b` to be assigned to `a`; the array itself isn't copied.

Array Arguments

- Although declaring a `parameter` to be an array is the same as declaring it to be a pointer, the same isn’t true for a `variable`.
- The following declaration causes the compiler to set aside space for 10 integers:
  ```c
  int a[10];
  ```
- The following declaration causes the compiler to allocate space for a pointer variable:
  ```c
  int *a;
  ```

Array Arguments

- A function with an array parameter can be passed an array “slice”—a sequence of consecutive elements.
- An example that applies `find_largest` to elements 5 through 14 of an array `b`:
  ```c
  largest = find_largest(b + 5, 10);
  ```
Pointers and Multidimensional Arrays

- C stores two-dimensional arrays in row-major order.
- Layout of an array with \( r \) rows:

\[
\begin{array}{ccc}
\text{row 0} & \text{row 1} & \text{row } r-1 \\
\vdots & \vdots & \vdots \\
\end{array}
\]

- If \( p \) initially points to the element in row 0, column 0, we can visit every element in the array by incrementing \( p \) repeatedly.

Processing the Elements of a Multidimensional Array

- Consider the problem of initializing all elements of the following array to zero:
  \[
  \text{int } a[\text{NUM_ROWS}][\text{NUM_COLS}];
  \]
- Using nested for loops:
  \[
  \text{int row, col;}
  \]
  \[
  \text{for (row = 0; row < \text{NUM_ROWS}; row++)}
  \]
  \[
  \text{for (col = 0; col < \text{NUM_COLS}; col++)}
  \]
  \[
  a[row][col] = 0;
  \]

Processing the Elements of a Multidimensional Array, cont’d

- If we view \( a \) as a one-dimensional array of integers, a single loop is sufficient:
  \[
  \text{int } *p;
  \]
  \[
  \text{\quad for (p = &a[0][0];}
  \]
  \[
  \quad p <= &a[\text{NUM_ROWS}-1][\text{NUM_COLS}-1]; p++)
  \]
  \[
  \quad *p = 0;
  \]

Processing the Rows of a Multidimensional Array

- A pointer variable \( p \) can also be used for processing the elements in just one row of a two-dimensional array.
- To visit the elements of row \( i \), we’d initialize \( p \) to point to element 0 in row \( i \) in the array \( a \):
  \[
  p = &a[i][0];
  \]
  or we could simply write
  \[
  p = a[i];
  \]

How String Are Stored

- Since a string is stored as an array of char, the compiler treats it as a pointer of type char *.
- Example:
  \[
  \text{char } p[] = \text{"abc"}; \quad \text{or}\]
  \[
  \text{char } *p;
  \]
  \[
  p = \text{"abc"};
  \]
- So, what is the type of the first argument of printf?

String Literals vs Character Constants

- A string literal containing a single character isn’t the same as a character constant.
  - "a" is represented by a pointer.
  - 'a' is represented by an integer.
- A legal call of printf:
  \[
  \text{printf("\n");}
  \]
- An illegal call:
  \[
  \text{printf('\n');} \quad /\ast\ast \text{WRONG} \ast\ast/\]

String Literals vs Character Constants

- A string literal containing a single character isn’t the same as a character constant.
  - "a" is represented by a pointer.
  - 'a' is represented by an integer.
- A legal call of printf:
  \[
  \text{printf("\n");}
  \]
- An illegal call:
  \[
  \text{printf('\n');} \quad /\ast\ast \text{WRONG} \ast\ast/\]
Character Arrays vs Character Pointers

- The declaration
  ```
  char date[] = "June 14";
  ```
declares `date` to be an array.
- The similar-looking
  ```
  char *date = "June 14";
  ```
declares `date` to be a pointer.

However, there are significant differences between the two versions of `date`.
- In the array version, the characters stored in `date` can be modified.
- In the pointer version, `date` points to a string literal that shouldn't be modified.

Q: How many bytes will be allocated for each case?

Character Arrays vs Character Pointers

- The declaration
  ```
  char *p;
  ```
does not allocate space for a string.
- Before we can use `p` as a string, it must point to an array of characters.
- One possibility is to make `p` point to a string variable:
  ```
  char str[STR_LEN+1], *p;
  p = str;
  ```
- Another possibility is to make `p` point to a dynamically allocated string. (Later)

Accessing the Characters in a String

- A version that uses pointer arithmetic instead of array subscripts:
  ```
  int count_spaces(char *s)
  {
    int count = 0;
    for (; *s != '\0'; s++)
      if (*s == ' ')
        count++;
    return count;
  }
  ```

Is it better to use array operations or pointer operations to access the characters in a string? We can use either or both. Traditionally, C programmers lean toward using pointer operations.

Should a string parameter be declared as an array or as a pointer? There's no difference between the two.